A Functional Structural Tree Model
LIGNUM
User's Guide & Reference Manual
Draft

Jari Perttunen

December 1, 2014

Contents

1 Introduction **3**

2 Three Dimensional Tree Models **5**
2.1 The Role of Three Dimensional Tree Models 5
2.2 Examples of Models 5
2.3 Applications 5

3 Programming Guidelines **6**
3.1 Background 6
3.2 Using C++ 7
3.2.1 C programmers 7
3.2.2 Learning C++ 8
3.3 Programming Style 9
3.3.1 Typographical Issues 9
3.3.2 Comments and Documentation 10

4 Getting Started with LIGNUM **12**
4.1 Getting Lignum 12
4.2 Compiling LIGNUM 12
4.3 Makefiles for LIGNUM 13
4.3.1 Predefined Macros 13
4.4 Running LIGNUM 14
4.4.1 Initialization of the Tree 14
4.5 Visualizing Results 15
4.5.1 The Visualization Program 15
4.5.2 MineSet Program 15

5 The Design of LIGNUM **16**
5.1 Units of LIGNUM 16
5.2 Program Architecture 18
5.3 Algorithms and Methods 20

6 Algorithms **22**
6.1 ForEach 22
6.2 Accumulate 23
6.3 AccumulateDown 25
6.4 PropagateUp 26

7 Classes in Lignum **28**
7.1 TreeCompartment 28
7.1.1 The Class Declaration 28
7.1.2 Functions an operations 29
7.1.3 Constructors 29
7.1.4 Destructor 30
7.1.5 Metabolic processes 30
7.1.6 Data Members 31
7.2 The Class Tree 31
7.2.1 The Class Declaration of Tree 31
7.2.2 Functions and Operations on the Class Tree 32
7.3 The Class Declaration of TreeAttributes 33
7.4 Public Data Members of TreeAttributes 33
7.5 The Class Declaration of TreeParameters 34
7.6 Public Data Members for TreeParameters 34
7.7 The Class TreeTransitVariables 35
7.8 The Class Declaration of TreeTransitVariables 36
7.9 Public Data Members of TreeTransitVariables 36

8 Units **37**

Chapter 1

Introduction

$\mathcal{LIGNUM}$ is a tree model developed by the researches in the Finnish Forest Research Institute, The University of Helsinki and The Helsinki University of Technology. The model $\mathcal{LIGNUM}$ has been designed to capture the three dimensional structure of a tree species and, at the same time, to be able to model its growth and development.

The three dimensional structure of a tree is captured with the help of simple structural units close to their real world counterparts. The metabolic functioning (photo synthesis, respiration, transport of nutrients) is modelled by associating appropriate functions to these structural units. The growth and senescence is modelled by changing the size of these units, creating new units and deleting these units during a simulation.

The software that implements $\mathcal{LIGNUM}$ has been programmed in C++. Since $\mathcal{LIGNUM}$ was first introduced [4, 3] a lot of things have happened. During the first implementations of the model the terminology was not established and this inevitably was reflected in the software. Also, the main tool the C++ programming language was still under development. Thirdly, as characterizing any research project many things were found by trial and error, by experimenting different alternatives. This too, was reflected in the software.

Today many things are different. The terminology used in $\mathcal{LIGNUM}$ is now established and approved by the scientific community. The C++ ANSI Committee has finished its work. The major addition to the language from the point of view of $\mathcal{LIGNUM}$ is the Standard Template Library (STL) now providing the ubiquitous abstract datatypes needed in software development. The research work is still going on, but we have now established a solid set of parameters, state variables and functions that can be used to describe a functioning of a tree with $\mathcal{LIGNUM}$. In addition, $\mathcal{LIGNUM}$ is no longer a research project *per se*, but also used by, e.g., the students of Helsinki University of Technology.

All this leads to only one conclusion: The time has come for a major rewrite of the software to keep The Tower of Babel from rising.

This document describes the C++ implementation of the tree model $\mathcal{LIGNUM}$. The classes, algorithms and other utilities are available in the library called libLGM.a and related C++ header files. The implementation uses the STL designed as part of ANSI C++ specification. In fact, if you understand the principles of STL-library it should be relatively easy for you to undestand

the design and implementation of $\mathcal{LIGNUM}$.

The software implementing $\mathcal{LIGNUM}$ is a result of an ongoing research work, thus only a momentary (a "snapshot") description can be given. The interested $\mathcal{LIGNUM}$ user is encouraged to report errors (both in the software and in the documentation) to the authors.

Chapter 2

Three Dimensional Tree Models

2.1 The Role of Three Dimensional Tree Models

The study of three dimensional tree models, also known as virtual trees, is an active field of research motivated by several reasons.

For the first, there is always the basic research interested in finding formal methods describing the architecture of trees and plants in general. One of the most succesful of these methods are Lindenmayer systems introduced by Aristid Lindenmeyer.

2.2 Examples of Models

The AMAP (Atelje pour Modelisation de l'Architecture des Plants) approach.

Lindenmayer systems.

Other methods to describe branching.

2.3 Applications

But the basic research always calls for applications.

1.Individual trees
-radiation, photosynthesis
-mechanics
-hydraulic modelling
-wood quality
(e.g, visualize the quality of wood from forest to end products)
2.Forest
-dynamics of (mixed species) stands
-forest damage
-matter exchange
3.Commercial Applications
-Entertainment industry
-Landscape Architecture

Chapter 3

Programming Guidelines

3.1 Background

The programming language used to implement Lignum is C++. Although it has bias and history in systems programming it is used by hundreds of thousands of programmers practically in every application domain. Some simple reasons can explain the popularity of C++. For the first it has language constructs that make it convenient (easy, efficient and reliable) to use many types of modern programming paradigms, most notably object oriented and generic programming. Secondly, it has language facilities that support the implementation of large (more than 1000 000 line) programs. It is possible to create alone small programs containing 1000 or so lines and make them work through brute force breaking each programming paradigm and rule of good style. But for any larger program requiring cooperation of several programmers this is simply not possible. You will notice that new errors are introduced more rapidly than the old ones are fixed. The program will become unmanageable.

Very few programs created today exist in one application area only. It is very common to have numeric applications with at least scientific visualization and user interaction but also data base access and perhaps even local networking. C++ was not designed for numeric applications but many numerical and scientific program libraries have been created in C++ simply because the necessary libraries for visualization and user interface already were implemented in C++.

Lignum is no exception. The basic data structures (classes) implementing a generic tree model don't easily fit into a e.g., Fortran mold. Further to be of any practical use at least scientific visualization must be provided with elementary user interface so that the results of the simulations can be studied. Lignum has also evolved from a small research experiment to a medium size project with many international connections. In the future also the programming work to develop the model for diverse purposes will be done by several independent teams of programmers. Hopefully the result of the programming work done in one team can be shared by other teams. Thus the maintenance of the program will be important. C++ has been a good choice for Lignum and due to the development of the language and the introduction of the Standard Template Library (STL) it is now also more convenient to use than ever.

3.2 Using C++

This is not intended to be a C++ programming manual. There are many excellent textbooks available. For example you may find [6] and [1] useful. Stroustrup's book not only is a complete tutorial and reference to the language but also gives a lot of practical programming advice how to use and not to use the language. Breymann's book teaches the use of STL, the essential tool for any serious C++ programmer.

Many of the developers of Lignum have and will have background other than computer science but they have some programming experience with C or Fortran. Here we have tried to collect some most common mistakes and pitfalls made by the novice C++ programmers and also try to give some elementary practical advice what to learn first to get started. You don't have to learn whole C++ to work with it. You can grow with the language.

3.2.1 C programmers

We have noticed that people with a strong C background tend to continue to write C with C++. It is possible to do so because C with a few exceptions is a subset of C++ but the potential benefits of the language are lost. Here are some common mistakes and suggestions what to do instead.

- Avoid macros. Use inline functions if you want to avoid function call overhead and for simple macros defining a constant consider const or enum constructs.
- Forget malloc and free. The new and delete operators do the same job better. Remember to use delete to avoid memory leaks. If you seem to need realloc consider if some predefined data structure for example vector fulfills your needs. In general avoid unnecessary use of new operator. Note that it is possible to create temporary variables, e.g., local to a function on the fly whenever you need one without using dynamic memory allocation. In fact only objects (like new tree parts) that may have a life time of the entire program need to be created dynamically with the new operator and stored in some container.
- Avoid pointers. Use references that are safer instead as formal parameters and return values. When you have a reference you know there is a real object you can deal with. With pointers to create safe code you always have to check if it is a NULL pointer or not. If you have to deal with the pointers yourself try to implement clean interface that provides references to the user.
- Forget C cast. If you need type cast learn to use dynamic, static and const casts in C++. They provide type checking.
- Use const. Everything that is constant (does not change) should be declared const. The more you program the more you'll learn to respect this little keyword. Do it from the beginning. It is practically impossible to correct afterwards.

- Avoid the use of built-in arrays and C style strings. There are predefined string and vector classes in the STL library. They simplify the programming and you don't loose in efficiency either. For numerical computations there is valarray class.

- If you need to use unions, to do complicated pointer arithmetic, bit shifting or some other exotic language feature hide the implementation deep in some function or class. Provide clear and easy to use interface. But when developing Lignum you are unlikely to do any of these. Lignum is a tree model and you are more likely to create data structures by refining existing ones to model tree organs and implement numerical methods to simulate their metabolic processes.

3.2.2 Learning C++

The reason to learn C++ is not to learn a new syntax to do things that has been previously done with Fortran or C. The reason is to learn to to design and implement systems better and more efficiently in a way C++ supports the work. This is like learning a new natural language; it will take years of practice to read or write for example French fluently and correctly as a foreign language.

It is not necessary to learn C first to learn C++. In fact it might be beneficial to learn C++ directly without any knowledge of C. As said it is not so much of learning a new programming language syntax but learning new design and programming paradigms. Thus the top down approach might be the best way.

First learn the key language concept class and its importance in data abstraction when modeling the problem domain. Learn the concepts of methods, friend functions, operator overloading and data members. Also make yourself clear what is meant by access control of class members. People coming from natural sciences may find easy to grasp the idea of inheritance that is simply hierarchical ordering of things to manage complexity.

Then expose yourself to STL that teaches you generic programming. The key concept in this paradigm is an abstract datatype and the generic algorithms that can be applied to that datatype. You will learn the important language concepts of templates and functors that are used extensively to implement STL. Also Lignum follows the paradigm of generic programming. The tree is implemented as an abstract datatype with a set of generic algorithms to apply various metabolic processes or collect information implemented as methods or functors to it.

All in all by learning first the high-level techniques of C++, then gradually the common subset of C and C++ you'll notice that there is no need or very little advantage to know or use the most obscure language features. After all they are mostly used by systems programmers implementing operating systems, device drivers etc. You will see that with C++ it is possible to apply good programming style to create applications that are easy to read and understand using program design based on sound methodology.

3.3 Programming Style

3.3.1 Typographical Issues

To aid reading and understanding programs written by others a few simple typographical conventions used in Lignum and advice about writing comments are outlined. First, a few lines of fictitious C++ code that provides an example of the things to consider.

```
//The purpose of the class AClass is to ...
typedef double METER;
typedef double KG;

class AClass{
friend METER AFriendFunction(AClass& obj);
public:
   AClass();
   METER firstMethodForTheClass(KG fm);
private:
   KG Wf;
   double rho;
};


//The purpose of the method is to ...
//For the physiological motivation of
//the algorithm see ...
METER AClass::firstMethodForTheClass(KG fm)
{
}
```

The C++ class names are capitalized. If the class name is a compound word each word is capitalized. The method names in a class start with a lower letter. If the method name is a compound word the each following word is capitalized. All (friend) functions are written as class names, i.e., each word is capitalized. User defined types are written in capital letters.

In Lignum we have noticed that to minimize the learning and memorizing process try to be consistent with the paper published or to be published when choosing names for variables. For example in a tree segment we need to have a state variable for foliage mass. In publications the symbol for the foliage mass is W_f. So the name of the attribute in the class TreeSegment should be `Wf`. If a Greek letter is used in a paper as a symbol for a state variable or a parameter write it out as the name of the class attribute. For example the symbol ρ is used as the wood density. So the name of the attribute should be `rho`. In general we use LaTeX style to write out the mathematical symbols.

Consistent use of indentation will ease the burden of reading and understanding of program code. There are several styles in use and there is no reason to prefer one over another. Selecting and learning a text editor like emacs the proper indentation will become a trivia and in addition to that emacs provides

many other useful features improving your productivity when editing program source files.

3.3.2 Comments and Documentation

Needless to say comments are an essential part of a good program. But remember misused comments can seriously affect the readability of the program. Three self-evident rules of thumb for writing comments are

- Comment is meaningful, i.e., the program code is not evident.
- Comment describes the piece of a program precisely.
- Comment is up to date.

Don't bother to explain trivial pieces of a program. Something written in the language itself (like `a = b + c`) does not need any comments. Such comments ("a becomes b plus c") just increase the amount of text reader has to go through. Describe the piece of the program unambiguously. Ambiguous and incomprehensible comments are no use at all. Too often when the program changes the comments are left untouched. Plain wrong comments are worse than no comments at all.

Writing good comments is as difficult as writing the program. Some general guidelines may help to nurture this art:

- Instead of explaining the non-trivial algorithm in detail you can refer to manuals, text books, publications etc.
- Comment each class member and constant shortly for its purpose. Comment on data is usually more useful than on algorithms.
- Try to answer the questions what and why rather than how.
- Use "formal comments" to describe preconditions, postconditions domain and range of a method or a function especially when some side effects will occur.

Ideally when you return to your program after a few weeks you should still understand it yourself. Tricky code should be rewritten rather than exhaustively commented. Finally, comments don't replace the program documentation. In short the documentation should describe:

- How to install and operate the software system.
- How to use the system. There is no so simple system that is obvious to its user.
- The design and the requirements of the system.
- The implementation and the maintenance of the system.

Software maintenance is a little bit misleading concept. Software does not need oil change, checking the tire pressure etc. like your car does. But the idea is that because software engineering is human activity and thus any nontrivial software product will contain errors you should aid tracking down these errors. You can do this by at least designing and documenting a test set for your software product and its components.

You can study more about software documentation from [5]. Ideally the documentation should run in parallel with the programming work. It will mean rewritings of the manual due to the changes in the software but like in any writing process the documentation will require many revisions before it is useful anyway.

Chapter 4

Getting Started with LIGNUM

4.1 Getting Lignum

The development of Lignum is managed by using CVS version control. CVS has become a popular system that allows several people simultaneously to develop a program. To get Lignum type:

```
prompt% cvs checkout c++adt
prompt% cvs checkout stl-lignum
prompt% cvs checkout Firmament
```

stl-lignum implements the core model of Lignum, i.e., the general representation of deciduous and coniferous trees. c++adt implements general useful classes used in Lignum. Firmament implements customizable standard overcast sky. If you must obtain the software over the network type on your machine:

```
prompt% cvs -d ¡user¿@magnolia.metla.fi checkout c++adt
prompt% cvs -d ¡user¿@magnolia.metla.fi checkout stl-lignum
prompt% cvs -d ¡user¿@magnolia.metla.fi checkout stl-lignum
```

Naturally you must have a user account on the machine that is the CVS repository. In the example the repository is assumed to be magnolia.metla.fi. For efficient use of CVS see the manual written by Cederqvist [2].

4.2 Compiling LIGNUM

Before the compilation the c++adt and the stl-lignum must be at the same directory level:

```
prompt% ls -F
Firmament/ c++adt/ stl-lignum/
```

The STL library must also be part of the C++ compiler system. To compile both the c++adt and the LGM libraries simply go to stl-lignum directory and type make:

```
prompt% cd stl-lignum
prompt% make
```

To compile the test set for various generic algorithms used in Lignum type

```
prompt% make algorithms
```

The main Makefile for the make utility to build the LGM library not only compiles the LGM library but also assumes the existence of c++adt directory and if needed recompiles the c++adt library.

4.3 Makefiles for LIGNUM

The make utility is used as an integral part of software development in Unix environment. A description file for the make utility called Makefile contains a set of macros and a set of specifications. Each specification consists of a target, optional prerequisites for the target and optional commands to be executed by the utility when a prerequisite is newer than the target.

4.3.1 Predefined Macros

There are several Makefiles that are used to compile the LGM and the c++adt libraries. To keep the libraries consistent (e.g., to compile for the 32 or 64 bit machine architectures) and to ease the change of a compiler some useful common macros throughout the Makefiles are predefined.

- CCC The compiler.
- CCCFLAGS The -c option must be present to tell the compiler to generate only object files (i.e., no linkage) from source files.
- OPTIMIZE Flags to optimize the object code.
- ARCOMMAND The program to create a program library.
- ARCOMMANDFLAGS Command line flags for the archive program.

For example on SGI to maximize the performance for 64 bit code for IP28 target platform using inter procedural analysis with default settings type:

```
prompt% make "OPTIMIZE=-Ofast=ip28 -IPA" algorithms
```

In general the the appropriate selection for a SGI platform can be determined by running "hinv -c processor". To study inter procedural analysis see manual page for ipa. To compile on Linux type:

```
prompt% make "CCC=g++" "CCCFLAGS=-c" "OPTIMIZE=-O"
"ARCOMMAND=ar" "ARCOMMANDFLAGS=rv" algorithms
```

There are some other useful Makefile targets defined.

- html Generate www documentations of the program files. The enscript program must be installed.

- changelog Generate ChangeLog file containing the summary of the cvs logs.
- clean Remove compiled files.
- distclean Remove compiled files also in c++adt directory.
- depend Create file dependencies for LGM library.

You may find the Makefile for LGM library useful when starting your own project.

4.4 Running LIGNUM

4.4.1 Initialization of the Tree

The parameters and functions of the tree for the simulation must be given in files. The file that is given as command line argument is the schema file i.e., the file that gives the names of files where the parameters and functions are defined, not the parameters and functions themselves. An example of the schema file is below:

```
#File describing the location of definition files
#for parameters and fuctions of LIGNUM
Parameters:
    Tree: Tree.txt
    Firmament: Firmament.txt
Functions:
    FoliageMortality:FoliageMortality.fn
    Buds: Buds.fn
    DoI: DoI.fn
```

The format of the schema file is simple. For the first, the `#` character starts a comment that extends to the end of line.

The schema file has two main sections, one for parameters and the second one for functions denoted by keywords `Parameters` and `Functions` followed by a colon.

The sections for parameters and functions consists pairs composed by a keyword,(e.g., `Tree` and `Foliage Mortality`) and a file name (`Tree.txt` and `FoliageMortality.fn` respectively). The keyword tells the purpose of the file, so that during the intialization parameters and functions are intialized properly. The colon is used to separate the keyword from the file name.

If necessary, the the sections for functions and parameters will be subdivided in the future.

An example of the parameter file for the tree is below.

```
#Parameters for tree compartments according to papers in
#Annals of Botany 1996 and in Ecological Modelling 1998.

af      1.30        #Needle mass-tree segment area (kg/m^2)
                    #relationship
```

```
ar      0.50        #Foliage - root relationship
lambda  1.3         #Intial value for lambda
lr      100.0       #Length - radius relationship of a
                    #tree segment
mf      0.20        #Maintenance respiration rate of foliage
mr      0.240       #Maintenance respiration rate of roots
ms      0.0240      #Maintenance respiration rate of sapwood
na      0.7854      #Needle angle (pi/4)
nl      0.10        #Needle length (10 cm = 0.10 m)
q       0.10        #Tree segment shortening factor
sr      0.330       #Senescence rate of roots
ss      0.07        #Senescence rate of sapwood
rho     400.0       #Density of wood in tree segment
pr      0.0010      #Proportion of bound solar radiation
                    #that is used in photosynthesis
xi      0.60        #Fraction of heartwood in tree segments
```

The file simply contains parameter value pairs. Each parameter name is reserverd keyword so that the program can recognize it. The `#` character begins a comment extending to the end of line.

The functions defining different behavior in the tree are given as parametric curves in ASCII files. See the class `ParametricCurve` in `libc++adt.a` for details.

The keywords `FoliageMortality`, `Buds`, and `DoI` denote functions for foliage mortality, number of new buds and relative shadiness (degree of interaction) respectively. More functions will be implemented if necessary.

Currently directory paths are not parsed so all the files, the schema file and the files defining the parameters and functions, must be in the same directory where $\mathcal{LIGNUM}$ is started.

4.5 Visualizing Results

4.5.1 The Visualization Program

4.5.2 MineSet Program

Chapter 5

The Design of LIGNUM

The design of any computer program includes the decision how to represent information and concepts from the real world. C++ supports many programming methodologies (perhaps most notably object orientation) but the emergence of the Standard Template Library emphasizes generic programming with abstract datatypes that encapsulates data and algorithms working with this data.

Two language constructs in C++ are of special importance for generic programming: classes and templates. A class defines a datatype consisting a set of possible states and operations defining transitions between those states. Template is simply the C++ term for a parameterized type.

5.1 Units of LIGNUM

The main design principle in $\mathcal{LIGNUM}$ is to model a tree with few simple basic units that correspond to the organs of the tree. Two main critearias must be met. For the first, the units must capture both the three dimensional structure of the tree and its metabolic processes. This is simply because structural development of a tree effects its metabolic processes and vice versa. For the second, the model must be computationally manageable to create forest stands consisting of different tree species. Ideally the units should be such that they can be divided into more detailed tree compartments if necessary.

Currently a tree is modeled in $\mathcal{LIGNUM}$ with five units called tree compartments. The two basic units are tree segment (TS) and bud (B). They are called basic units because they are not aggregates of other units. Axis (A) models one single branch in a tree. It is an aggregate or more precisely a sequence of tree segments and branching points (BP) ending with a terminating bud (Figure 5.1). The fifth unit is naturally the tree itself, an aggregate of all its tree compartments. The structure of $\mathcal{LIGNUM}$ has caused some confusion so let us make a exact unambiguous (recursive) definition.

Definition 1 ($\mathcal{LIGNUM}$) *The model tree $\mathcal{LIGNUM}$ consists of one axis. An axis is a possibly empty sequence of tree segments, branching points and one terminating bud. Each tree segment only must be followed by exactly one branching point. Branching point is a set of zero or more axes capturing the branching structure of the model tree.*

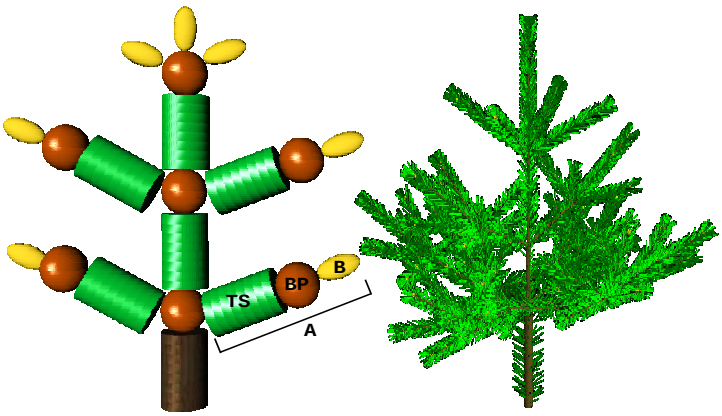

Figure 5.1: A model tree presented in $\mathcal{LIGNUM}$ (left). Cf. young Scots pine produced by $\mathcal{LIGNUM}$ (right). A = Axis, TS = Tree segment, BP = Branching Point, B = Bud.

The main functioning unit tree segment consists of sapwood, heartwood, bark and foliage and it captures the main metabolic functioning in a tree. The tree segment corresponds roughly to the annual shoot in trees but in the strict modeling sense this is not necessarily always the case, especially when modelling deciduous trees. Also annual rings are modeled. In real the annual rings are a result of differences in diameter growth in the spring and later in the autumn. In $\mathcal{LIGNUM}$ an annual ring is a number that is the radius without bark of the tree after the growing period. Thus to be precise the annual rings are not structural components in $\mathcal{LIGNUM}$ but merely a book keeping process. For coniferous trees all the structural components of the tree segment including foliage can be modeled as (hollow) cylinders (Figure 5.2) but for deciduous trees the foliage needs another approach. Clearly a more detailed description for the leaf is needed.

To start with one can identify two main structural components in a leaf: the leaf itself and the petiole that connects a leaf to a shoot. Leaves in deciduous trees come in all shapes and sizes but in $\mathcal{LIGNUM}$ the ellips is chosen to be the form of a model leaf. More precisely an individual leaf of a specific tree species is modeled with the smallest ellips that encloses the leaf. For some species this can be too rough of an approximation. The position of the petiole in space is defined by its starting and end points. The position of the leaf itself in space is defined by the end point of the petiole connecting to one of the axes of the leaf (ellips) and the leaf normal. One or more leaves can be connected to a tree segment. The tree segment for deciduous trees is in Figure 5.2.

The bud is an embryonic shoot, the growing point of the tree from which new shoots, leaves and flowers may develop. Buds can be divided into terminal buds located at the top of axes and lateral (or axillary) buds located in the tree segment. For deciduous trees the lateral buds are located where the petiole of a leaf is attached to a tree segment (Figure 5.2).

From the modeling point of view the branching point is the place where two

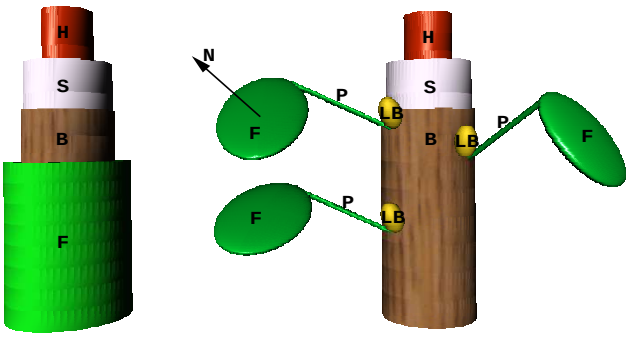

Figure 5.2: Tree segment for a coniferous tree (left) and for a deciduous tree (right). H = heartwood, S = sapwood, B = bark, LB = lateral bud, F = Foliage, P = petiole, N = leaf normal.

or more tree segments are connected to each other. The axis corresponds to a stem or a single branch of a tree.

5.2 Program Architecture

As a tree is represented with five units in $\mathcal{LIGNUM}$ a natural choice is to represent each unit as a class. Each unit will have its own set of attributes and operations. An axis being a dynamic sequence of alternating tree segments and branching points ending with a terminating bud can be easily represented by a list. A common notation for a list is to denote the beginning of a list with a left bracket ([) and the end of the list with a right bracket (]). The elements in the list are separated by commas (,). With this formal notation the the sample tree in Figure 5.1 can be now expressed as:

$$[TS, BP, TS, BP, TS, BP, B]$$

Branching point connects a set of axes in a tree. Thus a logical and consistent design choice is to define branching point as a list of axes. For example, a more detailed structural description of the sample tree folds out the branching points in the main stem. The two axes containing only terminating buds in the last (third) branching point are also folded out:

$$[TS, [A, A], TS, [A, A], [[B], [B]], B]$$

Further more the classes for the units of $\mathcal{LIGNUM}$ are organized in a hierarchy with a common (abstract) base class called TreeCompartment (Figure 5.3 on page 19). This design allows us to utilize polymorphism when implementing metabolic processes. Polymorphism means that a programming language allows operations of the same name (e.g., photosynthesis) to take different forms or implementations in different units.

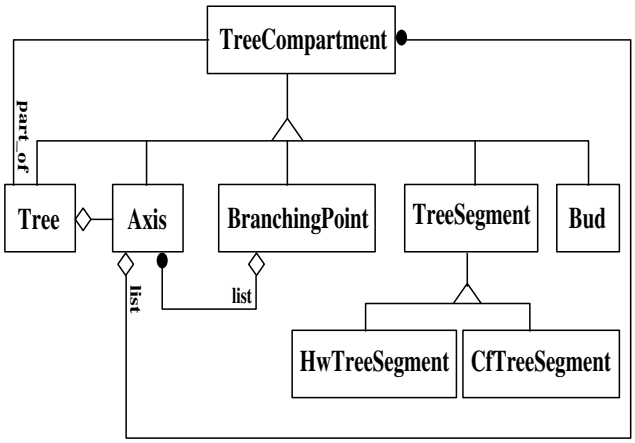

Figure 5.3: The class hierarchy in $\mathcal{LIGNUM}$ with OMT notation. Triangles and diamonds denote subclass relationship and aggregation respectively. Hw = Hardwood, Cf = Coniferous.

By defining attributes, methods and functions that describe different tree compartments one could start to implement different tree species. This is in fact how the first tree species, Scots pine, Jack pine and sugar maple were implemented. However, one of the problems with the software used to simulate these tree species is that it has proven to be inconvinient to modify and maintain.

Typically when creating and validating tree models with $\mathcal{LIGNUM}$ one needs to experiment with different parameter values, light models, metabolic processes etc. The reason why the program implementing $\mathcal{LIGNUM}$ has been troublesome to adapt for different tree species is that the classes defined (Tree, Axis etc.) are still concret data types.

To improve the flexibility of the software one more level of abstraction is needed. Instead of concrete datatypes the tree compartments must be defined as abstract datatypes. This means type parameterization of the tree compartments. One could naturally parameterize all tree compartments but so far the two most important functional tree compartments are the tree segment and the bud. The axis and the branch whorl are essentially containers and have not been subject to any modeling work per se.

The type parameterization of the tree segment and the bud defines a tree as an abstract datatype. Each tree species is an abstraction of the structure and function of tree segments and buds.

As an example suppose a modeler wants to study the development of Scots pine. The first thing to do is to create a subclass of the coniferous tree segment (CfTreeSegment, Figure 5.3) say, PinusSylvestrisTreeSegment. Then if the default methods implementing the metabolic processes are not satisfactory the modeler must redefine the appropriate methods. After the necessary changes in the class PinusSylvestrisTreeSegment a tree modeling a Scots pine can be spawned simply:

```
Tree<PinusSylvestrisTreeSegment> pinus;
```

If also the structure and functioning of the bud need changes when modeling for example the Scots pine a subclass of Bud , e.g., PinusSylvestrisBud, is required. After that a new model for the Scots pine can be brought into being:

```
Tree<PinusSylvestrisTreeSegment,PinusSylvestrisBud> pinus;
```

Similarly when modeling a birch the first thing to do is to create a subclass of deciduous tree segment (HwTreeSegment, Figure 5.3) say, BetulaPubescensTreeSegment. After the appropriate metabolic processes are redefined a tree modeling a birch can be set up:

```
Tree<BetulaPubescensTreeSegment> betula;
```

Once more one could argue that also axes and branching points should be abstracted too. But so far there has not been a need to assign metabolic processes to these tree compartments. However, it is possible due to so called default template parameters to add these tree compartments for the class Tree later without breaking the existing programming interface.

5.3 Algorithms and Methods

To complete the design of $\mathcal{LIGNUM}$ the necessery algorithms implemented either as generic functions or methods must be identified. Naturally one must accept the fact that as in any software development this process is incremental and ongoing work during the life time of the program.

The fundemantal decision is what should be assigned as methods for the tree compartments. Technically all algorithms could be implemented as methods but this is hardly a wise decision. It would only lead to constant changes to the classes identified so far making them difficult to maintain and increase the time to learn to use them. As tree compartments are created to describe the structure and function of a generic tree the design choice is that *only the metabolic processes* should be assigned as methods for the tree compartments. Other algorithms traversing the model tree passing information between tree compartments or quering the status of the tree should be implemented as generic functions.

To decide what actually are the metabolic processes depends on the level of detail how the tree is modeled. Scientist considering a tree growth on cell level views the tree differently as a scientist working on a forest level. $\mathcal{LIGNUM}$ models a tree capturing individual tree segments. Thus the metabolic processes that can be assigned to the tree compartments are photosynthesis, respiration, flow of water and nutrients, length growth, diameter growth and the mortality of tree compartments.

As the metabolic processes are assigned to individual tree compartments generic algorithms are needed to traverse the tree and apply these processes to each compartment. So far all metabolic processes can be implemented with one of the following four generic algorithms or as their combination: ForEach, Accumulate, AccumulateDown and PropagateUp. As the name suggests ForEach and Accumulate work as their counter parts in STL. AccumulateDown traverses the tree from the tip of the branches to the base of the tree. PropagetUp does the inverse. It traverses the tree from the base to the tip of the branches.

Finally not all computations are well suited or even possible within the structural description of a tree provided by $\mathcal{LIGNUM}$. For example assume that the model for photosynthesis is based on amount of photosynthetically active radiation (PAR) available for foliage. In $\mathcal{LIGNUM}$ one implementation to compute PAR is based on the pairwise comparison of tree segments to calculate first how they shade each other and then based on that the PAR can be solved. The complexity of such algorithm is $O(n^2)$. A better way could be to first create a (linear) algorithm using, e.g., Accumulate that collects the tree segments to a new data structure. After that with the help of some heuristics a better than $O(n^2)$ algorithm avoiding pairwise comparison of tree segments for the computation of PAR might be found.

Another example is the modeling work already being done. The flow of water can be modeled using so called connection matrix describing how the tree segments are connected. This information is also in $\mathcal{LIGNUM}$ and PropagateUp algorithm can be used to pass information upwards in a tree. But it always up to the modeler to decide the trade off between the work to adapt the computations to $\mathcal{LIGNUM}$ or to create the data structures from $\mathcal{LIGNUM}$ and use the familiar modeling frame work.

Chapter 6

Algorithms

So far all metabolic processes and queries of the status of the simulated tree can be implemented using one of the following four generic algorithms: ForEach, Accumulate, AccumulateDown and PropagateUp. Each of these traverse the tree in a slightly different manner and take care that the actual metabolic process or query implemented as a functor is applied to each tree compartment. Thus these algorithms eliminate the trouble of writing repetitive and error prone explicit loops each time a metabolic process or query is implemented or refined. The algorithms are linear, i.e., the algorithmic complexity is $O(n)$, where n is the number of tree compartments.

To use these generic algorithms include the file Algorithms.h into your program. Examples of these algorithms can be obtained by compiling and running the program `algorithms` in stl-lignum project directory. To compile the program type `make algorithms`.

The sample tree used in the examples consists of one main axis containing one tree segment, one branch whorl and the terminating bud. The branch whorl contains two auxilliary buds. Using the notation for the list the tree can be written as:

$$[TS, [[B], [B]], B]$$

Also the functors used in subsequent sections to clarify the use of the generic algorithms are the same as used in the sample program. The tree segment used in the examples is called MyTreeSegment and it is simply an subclass of TreeSegment.

6.1 ForEach

ForEach is similar to STL-library function for_each. In fact it applies for_each to each list of tree compartments in a tree. for_each in turn simply applies a user defined functor f to each tree compartment in a list.

Signature void ForEach(Tree¡TS¿& tree, const Function& f).

Arguments for the function.

tree The tree.

f The unary functor taking a pointer to a tree compartment as an argument. The functor must return the pointer to a tree compartment.

Returns The tree and tree compartments possibly modified by f.

The implementation of ForEach uses the notion function composition. That is, it defines functions h and g such that given the user defined function f:

$$h = (g \circ f)(x) = g(f(x))$$

The purpose of the function g is simply to traverse the tree. As an example consider the following functor DisplayType2 playing the role of the function f. The easiest way to ensure proper argument and return type for the functor is to inherit from AdaptableTCFunction¡TS¿. It is an empty class but predefines right argument and return types. It also defines (using STL class unary_function) necessary internal typedef-declarations.

```
template <class TS>
class DisplayType2: public AdaptableTCFunction<TS>{
public:
   TreeCompartment<TS>* operator ()(TreeCompartment<TS>* ts)const;
};
```

The implementation of the overloaded function operator simply prints out the type of the argument tree compartment.

```
template <class TS> TreeCompartment<TS>*
DisplayType2<TS>::operator()(TreeCompartment<TS>* tc)const
{
  if (Axis<TS>* myaxis =  dynamic_cast<Axis<TS>*>(tc)){
    cout << "Hello, I'm Axis" << endl;
  }
  //checking other tree compartments similarly
  else if ...
  ....
  return tc;
}
```

The call to ForEach to print out the type of each tree compartment is simply

```
ForEach(tree,DisplayType2<MyTreeSegment>());
```

ForEach algorithm can be used to implement metabolic processes e.g., photosynthesis and respiration or some other function like visualization where the computations on each tree compartment are inherently parallel. That is, tree compartments can be treated individually.

6.2 Accumulate

The algorithm Accumulate can be used to collect data, pass information from one tree compartment to another or query the status of a tree. It is similar to STL-library function accumulate. Note however, that in Accumulate the initial

value (also called the identity element) is passed and returned as reference, not by value as in accumulate. The reason for this design decision is that it is not always numerical data a modeller may want collect.

Accumulate traverses the tree and applies the user defined binary operator op to each tree compartment with the initial value that can be modified according to the binary operator.

Signature T& Accumulate(Tree¡TS¿& tree, T& init, const BinOp& op).

Arguments for the function.

tree The tree.

init The initial value. Also called the identity element.

op The binary operator. The functor must take two arguments: the initial value and the pointer to a tree compartment. The operator must return the modified initial value.

Returns The modified initial value.

More formally, the implementation of Accumulate uses the notion of function composition. It defines functions h and g such that given the user defined function f:

$$h = (g \circ f)(x, y) = g(f(x, y), y)$$

In practice the x is the identity and y is the tree compartment. The purpose of the function g is simply to traverse the tree and it does not modify the identity element or the tree compartment. As an example of the use of Accumulate consider the following binary operator CountCompartments:

```
template <class TS>
class CountCompartments{
public:
  int& operator ()(int& id,TreeCompartment<TS>* ts)const;
};
```

The implementation of the overloaded function operator in CountCompartments simply counts the number of tree compartments in a tree and echos the type of each compartment.

```
template <class TS>
int& CountCompartments<TS>::operator()(int& n,TreeCompartment<TS>* tc)const
{
  if (Axis<TS>* myaxis =  dynamic_cast<Axis<TS>*>(tc)){
    cout << "Hello, I'm Axis ";
  }
  //checking other tree compartments similarly
  else if ...
  ....
  n+=1;
  return n;
}
```

To count all tree compartments in a tree one simply calls Accumulate with the functor CountCompartments.

```
int i = 0;
int n = Accumulate(tree,i,CountCompartments<MyTreeSegment>());
```

The algorithm Accumulate can be used, e.g., to sum up the photosynthates and respiration of all tree segments in a tree. Another example could be a functor that collects all tree segments into a sequence. The computations *must not assume* in which order the tree compartments are traversed. Algorithms AccumulateDown and PropagateUp can be used to traverse the tree explicitely from top to bottom and from below upwards respectively.

6.3 AccumulateDown

The AccumulateDown algorithm can also be used to collect data, pass information from one tree compartment to anoter or query the status of a tree. However, as the name suggests the tree is traversed from the tip of the branches to the base of the tree. More over each axis is traversed independently (semantically in parallell). Note that the initial value is passed and returned as reference because the computations are not necessarily numerical.

AccumulateDown traverses the tree and applies the user defined binary operator op to each tree compartment with the initial value that can be modified according to the binary operator.

Signature T& AccumulateDown(Tree¡TS¿& tree,T& init, const BinOp& op).

Arguments for the function.

tree The tree.

init The initial value. Also called the identity element.

op The binary operator. The functor must take two arguments: the initial value and the pointer to a tree compartment. The operator must return the modified initial value.

Returns The modified initial value.

Because each axis is traversed independently the results of the compuations are summed up in branching points. Thus each initial element must have the overloaded add and assign (+=) operator defined with appropriate semantics. Also a new initial value is created for each axis so the call to a constructor must be possible. More precisely, if the type of the initial value is T the call to the constructor T() must succeed.

To count the number of tree segments with AccumulateDown and CountCompartments one simply writes:

```
int i = 0;
int n = AccumulateDown(tree,i,CountCompartments<MyTreeSegment>());
```

Assuming that the add and assign operator and the constructor are defined for the identity element[1] one can use AccumulateDown as Accumulate. However, some metabolic processes are depended on the information flowing downwards in a tree. These processes can be implemented only with the AccumulateDown algorithm. One such process is diameter growth as currently realized in $\mathcal{LIGNUM}$ (see [3]). Other examples can be various models for hormone and nutrient flows downwards in a tree.

6.4 PropagateUp

The algorithm PropagateUp can be thought as an inverse operation to AccumulateDown. The tree is traversed from the base to the tip of the axis. During the traversal the initial value is modified by the binary operarator. Further more each axis is computed independently, i.e., inherently in parallell. In each branching point a copy of the current value or status of the initial element is sent forward to each axis. Note again that the initial value is passed as reference. It is not always numerical data one wants to send up in a tree.

Signature void PropagateUp(Tree¡TS¿& tree,T& init, const BinOp& op).

Arguments for the function.

tree The tree.

init The initial value. Also called the identity element.

op The binary operator. The functor must take two arguments: the initial value and the pointer to a tree compartment. The operator must return the pointer to a tree compartment.

Returns Nothing.

Because each axis is computed independently a copy of the current value of the initial element is passed forward in a branching point. Thus the overloaded assignment operator (=) must be defined for the initial element.

As an example of simple signal passing in a tree consider the functor MyExampleSignal. It receives an signal as in integer value and at each tree compartment it increases signal's value by one.

```
template <class TS> TreeCompartment<TS>*
MyExampleSignal<TS>::operator()(int& n,TreeCompartment<TS>* tc)const
{
  if (Axis<TS>* myaxis =  dynamic_cast<Axis<TS>*>(tc)){
    cout << "Hello, I'm Axis ";
  }
  //checking other tree compartments similarly
  else if ...
  ....
  n+=1;
  return tc;
}
```

[1] Note that they are defined for the integer and floating types.

To pass a signal upwards in a tree starting from zero one simply calls PropagateUp with MyExampleSignal.

```
int s = 0;
PropagateUp(tree,s,MyExampleSignal<MyTreeSegment>());
```

Other possible applications for PropagateUp in addition to signal passing can be, e.g., modelling the water flow upwards in a tree.

Chapter 7

Classes in Lignum

7.1 TreeCompartment

The class TreeCompartment is the common base class for all tree compartments. It defines the position and the direction of a tree compartment but the main purpose of the class is to define a common set of metabolic processes implemented as virtual methods.

The metabolic processes can be further divided into two categories. The first category assumes that no flow of information from one tree compartment to another is required (thus also not allowed). This includes phosynthesis, respiration and mortality of tree compartments. The second category allows flow of information between tree compartments. These metabolic processes include length growth, diameter growth and flow of nutrients, water etc. in the tree.

The metabolic processes not allowing parameter passing assumes that all the information must be present in the tree compartment before the method is applied. Computationally these processes can be thought to be parallel. To apply these throughout the whole tree use ForEach algorithm using a predefined functor simply calling appropriate method. The metabolic processes allowing parameter passing can be applied throughout the whole tree by using either AccumulateDown or PropagateUp algorithms depending on the direction of the information flow in the tree. The actual type of the parameter implementing the flow of information between tree compartments is left to be decided in the user defined functor used in AccumulateDown or in PropagateUp.

The important design issue is weather the metabolic processes should be pure virtual or not. Conceptually the better solution would be to make them pure virtual. The class TreeCompartment is meant to be an abstract base class with no instances. But this could (and would) cause practical problems by requiring to implement each method at least as an empty method in each subclass. This is why the metabolic processes are predefined as empty in this base class.

7.1.1 The Class Declaration

```
template <class TS,class BUD>
class TreeCompartment{
  friend Point<METER> GetPoint(const TreeCompartment<TS,BUD>& tc);
  friend PositionVector GetDirection(const TreeCompartment<TS,BUD>&
```

```
tc);
  friend Tree<TS,BUD>& GetTree(const TreeCompartment<TS,BUD>& tc);
public:
  TreeCompartment();
  TreeCompartment(const Point<METER>& p, const PositionVector& d, Tree<TS,BUD>* t);
  virtual ~TreeCompartment();
  virtual void photosynthesis(){}
  virtual void respiration(){}
  virtual void mortality(){}
  template <class LGInOut> virtual void lengthGrowth(LGInOut& data){}
  template <class DGInOut> virtual void diameterGrowth(DGInOut& data){}
  template <class UpInOut> virtual void upFlow(UpInOut& data){}
  template <class DownInOut> virtual void downFlow(DownInOut& data){}
protected:
  Point<METER> point;
  PositionVector direction;
  Tree<TS,BUD>* tree;
};
```

7.1.2 Functions an operations

1. **GetPoint** Returns the position of the tree compartment.

   Signature friend Point¡METER¿ GetPoint(const TreeCompartment¡TS,BUD¿& tc).

2. **GetDirection** Returns the direction of the tree compartment.

   Signature friend PositionVector GetDirection(const TreeCompartment¡TS,BUD¿& tc).

3. **GetTree** Returns the tree the tree compartment belongs to.

   Signature friend Tree¡TS,BUD¿& GetTree(const TreeCompartment¡TS,BUD¿& tc).

7.1.3 Constructors

The default constructor predefines the position at (0,0,0) and the direction to be up, i.e., (0,0,1).

Signature 1 TreeCompartment().

The second constructor normalizes the direction to avoid subtile bugs when visualizing the tree with OpenGL graphics library.

Signature 2 TreeCompartment(const Point¡METER¿& p, const PositionVector& d, Tree¡TS,BUD¿* t)).

Arguments for the constructor.

p The position of the tree compartment.

d The direction of the tree compartment . Will be normalized.

t The tree the tree compartment is part of.

7.1.4 Destructor

The destructor is predefined empty but allows the use of polymorphism in the program.

7.1.5 Metabolic processes

The metabolic processes are predefined as empty methods but they designate a common interface that allows the use of polymorphism in the program.

1. Photosynthesis

   Signature virtual void photosynthesis().

2. Respiration

   Signature virtual void respiration().

3. Mortality

   Signature virtual void mortality().

4. Length Growth

   Signature virtual void lengthGrowth(LGInOut& data).

   Arguments to the method.

   data The data needed for the length growth, e.g., the λ. The type (LGInOut)of the argument is parameterized and must be defined in the functor used in the appropriate generic function PropagateUp, AccumulateDown etc.

5. Diameter Growth

   Signature virtual void diameterGrowth(DGInOut& data).

   Arguments to the method.

   data The data needed for the diameter growth. The type (DGInOut) is parameterized.

6. Up Flow

   Signature virtual void upFlow(UpInOut& data).

   Arguments to the method.

   data The data needed for the up flow The type (UpInOut) is parameterized.

7. Down Flow

   Signature virtual void downFlow(DownInOut& data).

   Arguments to the method.

   data The data needed for the down flow The type (DownInOut) is parameterized.

7.1.6 Data Members

1. **Point¡METER¿ p** The position of the tree compartment.
2. **PositionVector d** The direction of the tree compartment.
3. **Tree¡TS,BUD¿* t** The tree tree compartment belongs to.

7.2 The Class Tree

The class `Tree` captures the structure and functioning of a single tree. It also allows the user of the class to query the status of the tree and individual tree compartments (e.g., after a simulation to to collect data for further analysis).

The `Axis` is the list of tree compartments. The adjustable parameters defining the functionig of the different tree compartments are collected into class `TreeParameters` and the attributes describing the status of the whole tree during simulations are in the class `TreeAttributes`.

The third group of variables, called transit variables, are in the class called `TreeTransitVariables`. Transit variables are "technical". They don't have a biological meaning but are needed to proceed one interval of time (one time step) in a simulation. A transit varible may be needed for example to find a root of a function.

The class `TreeFunctions` collects user defined functions needed in modelling growth and senescence processes of the tree. Each function is defined as a parametric curve in an ASCII file. For the details, see the class `ParametricCurve` in the library `libc++adt.a`.

7.2.1 The Class Declaration of Tree

```
class Tree: public TreeCompartment{
friend void InitializeTree(Tree& tree,
                           const CString& schema_file);
friend TP GetAttributeValue(const Tree& tree, const TAD name);
friend YEAR GetAttributeValue(const Tree& tree, const TAI name);
template <class T1,class T2>
friend T2 SetAttributeValue(Tree& tree, const T1 name,
                            const T2 value);
friend TP GetParameterValue(const Tree& tree, const TPD name);
friend TP SetParameterValue(Tree& tree, const TPD  name,
                            const TP value);
friend TP GetTransitVariableValue(const Tree& tree,
                                  const TTD name);
friend TP SetTransitVariableValue(Tree& tree, const TTD name,
                                  const TP value);
public:
  Tree();
  Tree(const Point<METER>& p, const PositionVector& d);
private:
  TreeFunctions tf;
  TreeParameters tp;
```

```
  TreeTransitVariables ttp;
  Axis axis;
  RootSystem rs;
};
```

7.2.2 Functions and Operations on the Class Tree

The functions and operations on the tree are used to intiliaze the tree in the beginning of a simulation, querying and setting values for parameters, attributes and transit variables.

To query or set a value for a parameter (attribute or transit variable) with the functions available, the name of the parameter (attribute or transit variable) must be known. To maintain the consistency in the program, the name given as an argument to the function is a symbolic constant of the same name as the parameter in question.

For example, the following scenario can be used to check to query if the parameter value for tree segment - needle area relationship (field `af` in the class `TreeParameters`) is correctly installed.

```
Tree tree;
TP af1;
InitializeTree(tree,''SchemaFile'');
af1 = GetParameterValue(tree,af);
```

The call `GetParameterValue(t,af)` returns the value of the parameter.

The functions are overloaded and the types parameterized if necessary to maintain the transparency with different datatypes.

friend void InitializeTree(Tree& tree, const CString& schema_file);

Initialize the tree by reading parameter values and functions from in files found in `schema_file`.

Arguments for the function.

tree The tree.

schema_file The schema file naming the files (and their locations) for parameters and functions needed for simulation.

Returns Nothing.

GetAttributeValue();

Overloaded function to query the value of an attribute for the tree (one of the variable fields in the class `TreeAttributes`).

friend TP GetAttributeValue(const Tree& tree, const TAD name);

Arguments for the function.

tree The tree.

name The name of the attribute.

Possible values One of `M, P` or `Wr`.

Returns The value of the parameter. The type is is a symbolic type definition for built in double data type.

friend YEAR GetAttributeValue(const Tree& tree, const TAI name);

Arguments for the function.

tree The tree.

name The name of the attribute.

Possible values `age`.

Returns The value of the parameter. The type is is a symbolic type definition for built in int data type.

To query the age of the tree there one simply types `GetAttributeValue(t,age)`. The returned value is the age of the tree and the type is a symbolic type definition for built in int data type.

7.3 The Class Declaration of TreeAttributes

The class `TreeAtrtributes` collects attributes (state variables) of the tree. Eventhough the class is used as a private data member in the class `Tree` , access to attributes in a controlled manner is provided by using overloaded function `GetAttributeValue()` declared in the class `Tree`.

```
class TreeAttributes{
public:
  TreeAttributes();
  YEAR age;
  METER lb;
  KGC P;
  KGC M;
  KGC Wr;
};
```

7.4 Public Data Members of TreeAttributes

age Age of the tree in years.

Unit –.

lb Longest branch as a vertical projection from the main stem.

Unit m.

P Annual photosynthesis of the whole tree.

Unit kgC.

M Annual respiration of tree.

Unit kgC.

Wr Root mass of the tree.

Unit kgC.

7.5 The Class Declaration of TreeParameters

The class `TreeParameters` collects adjustable parameters for all tree compartments. It is a private members of the class `Tree` *but access to the parameters in a controlled manner is providided by using overloaded function* `GetParameterValue()` declared in the class `Tree`.

```
class TreeParameters{
public:
  TreeParameters();
  TP af;
  TP ar;
  TP lr;
  TP mf;
  TP mr;
  TP ms;
  TP pr;
  TP q;
  TP sr;
  TP ss;
  TP rho;
  TP xi;

};
```

7.6 Public Data Members for TreeParameters

af Needle mass - tree segment area relationship.

Unit kgm^{-2}.

Default value 1.3.

ar Foliage - root relationship.

Unit $kgkg^{-1}$.

Default value 0.5.

lr Length -Radius ratio for a new tree segment.

Unit –.

Default value 1.3.

mf Maintenance respiration rate of foliage.

Unit $kgCkgC^{-1}year^{-1}$.

Default value 0.2.

mr Maintenance respiration rate of roots.

Unit $kgCkgC^{-1}year^{-1}$.

Default value 0.24.

ms Maintenance respiration rate of sapwood.

Unit $kgCkgC^{-1}year^{-1}$.

Default value 0.024.

pr Propotion of *absorbed* (bound) solar radiation used in photosynthesis

Unit –.

Default value 0.001.

q Tree segment shortening factor.

Unit –.

Default value 0.1.

sr Senescence rate of roots.

Unit $1 \times year^{-1}$.

Default value 0.33.

ss Senescence rate of sapwood.

Unit $1 \times year^{-1}$.

Default value 0.07.

rho Density of wood.

Unit kgm^{-3}.

Default value 400.

xi Fraction of heartwood in new tree segments.

Unit –.

Default value 0.6.

7.7 The Class TreeTransitVariables

By transit variables of the tree it is variables that don't directly affect the behaviour and growth of the tree but are necessary to make computations to proceed one interval of time (one time step in simulation). More formally, let us denote the status of the tree at one moment of time t with $x(t)$, the functions and parameters for external conditions (e.g., weather) etc. needed to describe the behaviour of the tree with $u(t)$ and the transit variables with θ_i then we

can write $f(x(t), \theta_1 \ldots \theta_n, u(t)) \rightarrow g(x(t)) \rightarrow x(t+1)$ where f is the model (implemented e.g. as a computer program).

The class `TreeTransitVariables` is declared as a private data member in the class `Tree` *but the controlled access is provided using the function* `GetTransitVariableValue()` declared in the class `Tree` .

7.8 The Class Declaration of TreeTransitVariables

```
class TreeTransitVariables{
public:
  TreeTransitVariables();
  TP lambda;
};
```

7.9 Public Data Members of TreeTransitVariables

lambda Variable to balance carbon balance equation.

Unit –.

Default value 1.3.

Chapter 8

Units

The different units used in LIGNUM are defined in `LGMUnits.h`. In general, all dimensions are in meters and weigths in kilograms dry weight.

Name	Basic data type	Description
YEAR	unsigned int	Age of a tree (or tree compartment)
KGC	double	Kilogram of carbon (dry weight)
METER	double	Meter (used in dimensions of tree compartments)
TP	double	Adjustable parameters for tree (see class TreeParameter)

Bibliography

[1] Ulrich Breymann. *Designing Components with the C++ STL. A New Apporoach to Programming.* Addison-Wesley, Harlow, England, 1998.

[2] Per Cederqvist. *Version Management with CVS.*

[3] J. Perttunen, R. Sievänen, E. Nikinmaa, H. Salminen, H. Saarenmaa, and J. Väkevä. LIGNUM: A tree model based on simple structural units. *Annals of Botany*, 77:87–98, 1996.

[4] H. Salminen, H. Saarenmaa, J. Perttunen, R. Sievänen, E. Nikinmaa, and J. Väkevä. Modelling trees with object oriented scheme. *Math. Comput. Model*, 20(8):49–64, 1994.

[5] Ian Sommerville. *Software Engineering.* International Computer Science Series. Addison-Wesley, Reading, Massachusetts, second edition, 1985.

[6] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, third edition, 1997.